HONEYWELL

NEW USERS' INTRODUCTION TO MULTICS - PART II

SOFTWARE

SUBJECT

Introduction to Multics

SPECIAL INSTRUCTIONS

This manual is part of a new two-volume set entitled *New Users' Introduction to Multics* (Order No's. CH24 and CH25). The introductory set, along with one of the Multics text editor user guides, is the prerequisite to all further Multics manuals. The text editor user guides are:

qedx Text Editor Users' Guide	Order No. CG40
Emacs Text Editor Users' Guide	Order No. CH27

SOFTWARE SUPPORTED

Multics Software Release 8.0

ORDER NUMBER

CH25-00

November 1979

Honeywell

PREFACE

The purpose of this manual is to provide new users with an introduction to Multics use and a workbook that develops detailed applications from principles introduced in Part I of the New Users' Introduction to Multics (Order No. CH24). The topics covered here have been chosen either because they are useful and, with the introduction in Part I, comprehensible to the new user or because they illustrate fundamental elements of the Multics system.

The information presented here is a subset of that contained in the primary Multics reference document, the Multics Programmers' Manual (MPM). The MPM should be used as a reference to Multics once the user has become familiar with this introductory guide. The MPM consists of the following individual manuals:

Reference Guide	(Order No. AG91)
Commands and Active Functions	(Order No. AG92)
Subroutines	(Order No. AG93)
Subsystem Writers' Guide	(Order No. AK82)
Peripheral Input/Output	(Order No. AX49)
Communications Input/Output	(Order No. CC92)

Throughout this manual, references are made to both the MPM Commands and Active Functions manual and the New Users' Introduction to Multics Part I. For convenience, these references will be as follows:

MPM Commands
Part I

The Multics operating system is referred to in this manual as either "Multics" or "the system." The term "computer" refers to the hardware on which the operating system resides.

The information and specifications in this document are subject to change without notice. Consult your Honeywell Marketing Representative for product or service availability.

©Honeywell Information Systems Inc., 1979

File No.: 1L13

CH25-00

CONTENTS

		Page
Section 1	Introduction.	1-1
	Manual Conventions	1-1
	Peripherals.	1-2
	Terminals	1-2
	Card Readers.	1-2
	Storage Devices	1-3
	Line Printers	1-3
	Central Processing Unit (CPU).	1-3
	Input/Output Multiplexer (IOM)	1-4
	Front-End Network Processor (FNP). . .	1-4
	Computer Languages	1-6
Section 2	Multics Command Language.	2-1
	System and User-Written Commands . . .	2-2
	Commands Applied	2-2
	Multiple Commands.	2-5
	Reserved Characters and Quoted Strings	2-6
	Iteration.	2-8
Section 3	Active Functions.	3-1
	Active Functions as Substrings	3-2
	Nesting Active Functions.	3-3
	Iteration of Active Functions . . .	3-4
	Rescanning.	3-5
	Active Function Errors	3-7
Section 4	Important Command Language Features . . .	4-1
	Star Names	4-1
	Equal Names.	4-5
	Concatenation.	4-8
	How Commands Can Be Interrupted. . . .	4-8
Section 5	Abbreviation and Argument Substitution. .	5-1
	The abbrev Command	5-1
	The do Command	5-5
Section 6	exec_com.	6-1
	Creating an exec_com Segment	6-1
	Argument Substitution	6-2
	Control Statements.	6-3
	start_up.ec	6-7

		Page
Section 7	Additional Concepts	7-1
	Online	7-1
	Absentee	7-2
	Storage System	7-2
	Search Rules	7-5
	Linking	7-5
	Bound Segments	7-6
	Archive Segments	7-6
	Editor Macro	7-6
Appendix A	Glossary	A-1
Appendix B	Functional Breakdown of Selected Multics Commands	B-1
	Selected Commands Listed by Function	B-1
	Access to the System	B-1
	Storage System, Creating and Editing Segments	B-1
	Storage System, Segment Manipulation	B-2
	Storage System, Directory Manipulation	B-2
	Storage System, Access Control	B-2
	Storage System, Address Space Control	B-3
	Command Level Environment	B-3
	Communication Among Users	B-4
	Communication with the System	B-4
	Control of Absentee Computations	B-4
	Wordprocessing	B-5
Appendix C	Functional Breakdown of Selected Active Functions	C-1
	Reference to Active Function by Groups	C-1
	Arithmetic	C-1
	Character String	C-2
	Condition Handling	C-2
	Date and Time	C-2
	Logical	C-2
	Miscellaneous	C-3
	Pathname Manipulation	C-3
	Question Asking	C-3
	Storage System Names	C-3
	User/Process Information	C-4

Page

ILLUSTRATIONS

Figure 1-1. Components of Multics Hardware. 1-5
Figure 7-1. Hierarchical Storage System 7-4

SECTION 1

INTRODUCTION

MANUAL CONVENTIONS

Part II of the New Users' Introduction to Multics continues the discussion of Multics command language which was begun in Part I (CH24). Its purpose is to explain some of the detailed uses possible with the basic components of the language--commands and arguments--and to equip the user with a wide range of features and conventions that make command language extremely flexible and easy to use. This is accomplished through sections that illustrate detailed command applications, explain the use of active functions, and present important command language features such as the star convention and exec_com. Also included is a section on important computer concepts and their specialized application in Multics. Finally, there is a list of additional glossary items that supplement those in Part I and that will be valuable as you become more fully acquainted with computers and, in particular, Multics.

The conventions and special symbols used in this manual are the same as those in Part I.

Technical or other unfamiliar terms are underlined when used the first time and are included in the glossary (Appendix A).

When a command is referred to for the first time, its short name is shown in parentheses immediately following the long name. For example, print (pr).

Examples of lines printed on a terminal use exclamation points to indicate lines that the user types. These characters will not be typed by the system as a prompt to you, and they are not to be typed by you.

The ready message used in examples is the regular message printed by the system:

```
r 13:02 1.423 77
```

The first set of numbers tells the time of day on the 24-hour clock. The 13:02 indicates it is two minutes after one o'clock in the afternoon. The second set of numbers shows the amount of CPU time you've used since the last ready message, and the third number (77 here) indicates the number of pages of information brought into main memory from secondary storage since the last ready message. (See below for a discussion of CPU, memory and storage.)

But before we embark on this further discussion of Multics software, it may be interesting to you to understand the fundamental operation of computer hardware. Hardware is a term used to refer to the physical units that make up a computer system, the apparatus as opposed to the programs. Multics itself is not a computer; it is a software system, a system of sophisticated programs. As we shall see in the following discussion, there are a number of machines that comprise the hardware system that the Multics software runs on.

PERIPHERALS

Peripherals are machines that can be operated under computer control but do not perform the control and computational functions of the central computer. Terminals are peripheral devices with which you are probably already familiar. Others include keypunch machines and card readers, line printers, and storage devices such as magnetic tapes and disks.

Terminals

There are basically two types of terminals: printing or hardcopy terminals and video terminals, commonly called CRTs (cathode ray tube). Both have keyboards that resemble those on typewriters, and both accept output as well as input. The video screen shows your input and output on a television-like screen whereas hardcopy terminals print input and output on paper as you work and thereby provide immediately a printed record of your terminal session.

Card Readers

Card readers transfer programs and data punched on computer cards to the central computer. A necessary accompanying device is the keypunch, a typewriter-like machine with which you type characters onto computer cards.

Storage Devices

On most systems, storage consists of two parts--main memory and secondary storage. When information in secondary storage is to be processed, it is brought into main memory where it can be manipulated more rapidly. On the Multics system, however, all information can be processed at the same high speed, so there is no essential difference between main memory and secondary storage. This is the special Multics feature called virtual memory, which is discussed in Section 7.

The principal device used for storage in the Multics system is the disk; all information stored in the virtual memory system is on disks. Information can be stored on tapes and cards in the Multics system, but when it is, it is not part of the virtual memory system. A special connection must be made in order to process that information.

Line Printers

Line printers are so called because they print out results from a computer one line at a time. Unlike terminals, line printers are strictly output devices; they do not accept input. Line printers do, however, print output at very high speeds, much faster than terminals.

CENTRAL PROCESSING UNIT (CPU)

The central processing unit is the nerve center of the computer system; it coordinates and controls the activities of all the other units and performs all the arithmetic and logical processes to be applied to data. We can consider it as being divided into three functional parts:

1. an appending unit (APU)
2. a control unit
3. computation units

The appending unit locates information in memory and checks the access that the particular user has to the information. Multics software uses the APU to provide the unique level of security that protects information kept on the system.

The control unit acts as a synchronizer. It interprets instructions sent to it by the APU and issues the appropriate commands to the computation units.

There are two units in the CPU that perform computations: the operations unit and the decimal unit. Between them these two units perform basic computations such as addition, subtraction, division and comparison.

INPUT/OUTPUT MULTIPLEXER (IOM)

The input/output multiplexer processes information coming from terminals, card readers, and some parts of storage and returning to terminals or going to line printers. On some systems, a similar mechanism is called the input/output processor and sometimes it even resides within the CPU.

FRONT-END NETWORK PROCESSOR (FNP)

The front-end network processor is the piece of hardware incorporated into some systems to process information coming into and going out of the system by way of communication channels (e.g., telephone lines). Thus all input from terminals, and output directed to terminals, goes through the FNP because the terminals are all connected to the computer by one type of communication channel or another. Card readers, on the other hand, do not necessarily send their input through the FNP; they do so only if they are connected to the computer by way of communication channels, as is the case when they are at a different site than the computer. The FNP must be at the same site as the central computer.

With the description of major hardware items now complete, we can construct an illustration of how the hardware used by the Multics system is interconnected:

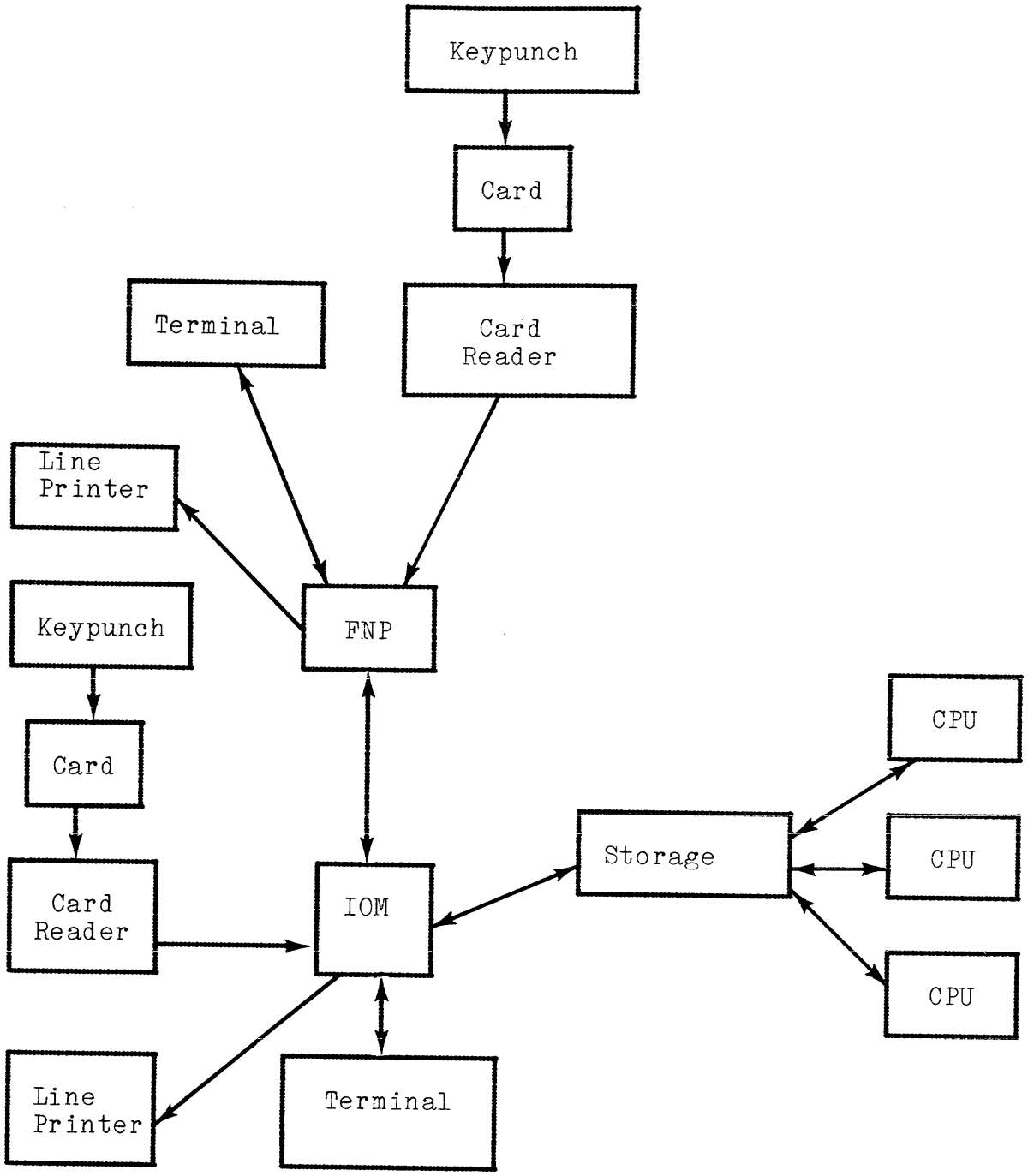


Figure 1-1. Components of Multics Hardware

COMPUTER LANGUAGES

The CPU and front-end processor operate according to a coded language that is very intricate and hard for humans to use efficiently. When we put information into the computer, we use coded languages that are easy for us to understand. These languages are usually called higher level languages. Some that you've probably heard of already are FORTRAN, PL/I, and COBOL. But these languages must be translated into the machine's language before the computer can perform its work. A compiler is the thing that translates a higher level language into machine language.

The compiler is not a hardware item in the computer system; it is a program. It resides in storage and can be called when needed. As a matter of fact, there are usually several compilers with each computer because each higher level language has its own program to translate it into the machine's code. On the Multics system there are compilers for FORTRAN, PL/I, COBOL, BASIC, and APL.

There is also another translating program similar to a compiler called an assembler. It too translates a programming language into machine code. But instead of translating a higher level language like COBOL, it translates a programming language called assembly, a language whose code is a symbolic form of machine language that resembles English much less than higher level languages do.

But the commands, active functions, texts, etc. that you enter at a terminal are not written in either higher level languages or assembly language. Instead, you are typing in a language (much of it specific to Multics) that calls or activates programs that are already written and reside in the system. On Multics, these programs are written in PL/I, and, indeed, these programs are the system, the Multics software system. What you type at a terminal calls programs and supplies data for their running, and it is to this procedure that we will now turn our attention for the remainder of this manual.

SECTION 2

MULTICS COMMAND LANGUAGE

A command is, as the name implies, a directive that you the user give the Multics system to make the system perform some action. You issue a command by typing its name at the terminal, along with arguments and control arguments, and concluding with a newline, a combination of a carriage return and linefeed. This sends the particular command message, or command invocation, to the command processor where it is evaluated and acted upon. If in evaluation the command processor finds that the line is improperly typed (which, in most cases, means misspelled), or if the command program finds that the line is incorrectly structured with arguments and control arguments, an error message is returned, indicating where the mistake is:

```
! primt report
  Segment primt not found.
  r 9:37 .144 55
```

```
! print 3 12 report
  print:  Entry not found. >user_dir_dir>Pubs>Smith>3
  r 9:37 .144 55
```

The command processor return the error message in the first example because it cannot locate a command with the misspelled name. In the second example, the command processor can find the command named "print," but the command program cannot find a segment whose pathname is "3." Thus the command cannot execute and so returns the error message. The print command will print part of a segment when line numbers are specified (e.g., 3 12); but in this case, the line designation comes incorrectly before the pathname of the segment to be printed. The correct syntax is:

```
! print report 3 12
```

When the processor and the command program find that all is well with the command invocation (which, by the way, is referred to simply as a command line throughout most of Multics documentation), then the command is executed. Finding that the invocation is correct means that the command processor was able to locate a program referred to by the command name given and that the command is capable of running on the arguments and control arguments given.

SYSTEM AND USER-WRITTEN COMMANDS

Most command programs in Multics are available throughout the entire system and can be executed by any user. There are, though, a few commands that can be used only by the system administrator, commands that control system usage, such as those needed to put new users on the system.

Then too, you the user can write special commands, normally called user-written commands, that can be executed in the same manner as system commands, though only by the user who creates them and anyone given access to them.

COMMANDS APPLIED

There is quite a large number of commands on the Multics system and these commands can be applied to a variety of situations. Most commands are adapted to particular situations by using arguments and control arguments, as is discussed in Part I of the New Users' Introduction. Some commands can operate by themselves, that is, without any arguments specified by the user. Commands of this type may not accept arguments, such as the command that prints the current working directory (print_wdir), or they may operate with certain preestablished arguments, default arguments, unless the user specifies otherwise. A good example of the latter is the help command discussed in Part I. Invoked without any control arguments, the help command prints information explaining how to use the help command. Then of course with the name of an info segment given as an argument, the help command will print the explanation contained in that info segment.

Another such command is list (ls), and it serves as a good example of how a command can be adapted to particular situations. When invoked without any pathname or control arguments, the list command prints the names of all segments in your working directory. For example:

```
!  ls

   Segments = 8,    Lengths = 41.

   r w    10  seg_1
   rew     9  seg_2
   r w     3  Smith.profile
   r w     7  work.list
   re      2  work
   r w     1  print.ec
   r w     1  output_file
   r       8  data_base
```

If, however, you only need to know how many segments exist in your working directory and how long they are, you can specify that the command return just the total shown in the heading by using the control argument -total (-tt):

```
!  ls -tt

   Segments = 8, Lengths = 41.
```

You can make this command return even more specific information by adding the pathname of a particular segment. You could, for instance, check the length of a particular segment by typing:

```
!  ls Smith.profile

   Segments = 1, Lengths = 1.

   r w    1  Smith.profile
```

You could also limit the amount of information printed about the segment. Suppose that the segment is known by several different names and you want to know only the names of the segment. By adding the -name control argument, you can get a list of the heading and the names without the access listing:

```
!  ls Smith.profile -name

   Segments = 1, Lengths = 1.

   Smith.profile
     John
     prof
```

Using this control argument can save processing time when you are listing names of a lot of segments, as you can do quite easily with the star convention discussed in Section 4.

In yet another case, you could check the last time the segment was altered by adding the date_time_contents_modified control argument (-dtcm):

```
!  ls  Smith.profile -name -dtcm

   Segments = 1, Lengths = 1.

   07/05/79   1456.3   Smith.profile
                       John
                       prof
```

The time of day is represented here in terms of the 24-hour clock; 1456.3 is 2:56 p.m. Knowing when a segment was last changed can be helpful in a variety of ways, such as keeping track of updates to the segment.

In most cases, especially when checking a specific segment, you don't need the header information, so that can be eliminated by typing is another control argument:

```
!  ls Smith.profile -name -dtcm -no_header

   07/05/79   1456.3   Smith.profile
                       John
                       prof
```

Control arguments give you a great of deal control over how comments execute. Normally control arguments follow pathname arguments on the command line, though most commands don't require that you arrange the control arguments in any particular order. For

instance, the three control arguments in the above command line could be arranged in any order after the pathname. And as is demonstrated below, control arguments can themselves take arguments that further specify how a particular command is to execute.

MULTIPLE COMMANDS

Sometimes it is easier to send several commands to the command processor at one time rather than send them separately. To do this you simply place a semicolon at the end of each command, after its arguments:

```
!  print_messages; print_wdir; who; help sked
```

Note: There is no need to type a semicolon after the last command and its arguments

If you were to send the commands separately, as we've been doing in examples up to this point, you would have to wait for the command to process and for another ready message to be printed before typing the next command line:

```
   r 10:31 0.662 07

!  print_messages
      .
      .
      .
   r 10:32 0.062 0

!  print_wdir
      .
      .
      .
   r 10:34 0.557 17

!  who
      .
      .
      .
   r 10:36 0.379 16

!  help sked
      .
      .
      .
   r 10:37 0.380 77
```

Note: The three dots between commands and ready messages are meant to represent the output from the commands.

But none of the four commands in this example require that you wait for the others to process; for instance, you needn't see the results of the print_messages command before typing the print_wdir command. So you can send them all to the command processor at the same time.

This technique is also useful when you are invoking commands that don't return information to your terminal. For example, when you change working directories, there is no reason why you couldn't send the next command to the command processor right away:

```
! change_wdir >udd>Pubs>Jones; print seg_2
```

It would be slower to type the commands on separate lines because you'd have to wait for another ready message before typing the second command. By putting both commands on the same line you will be ready, when the next ready message appears, to proceed with what you've learned by reading the contents of seg_2.

RESERVED CHARACTERS AND QUOTED STRINGS

The Multics command language reserves some characters to which special significance is attached. The reserved characters are: space, quotation mark ("), semicolon (;), the newline character, the vertical bar (|), parentheses, and brackets ([]). Earlier in this section we discussed special meanings of the semicolon and newline character. Here we will cover some special uses for the blank space and the quotation mark, and you will see the special significance of parentheses when we discuss iteration below. The special uses of brackets and the vertical bar are discussed in Section 3.

The space character is reserved for separating arguments, including command names, on the command line. For that reason character strings cannot contain blank spaces and instead simulate blanks with the underscore character, as is discussed in Section 3 of Part I.

Quotation marks are reserved for passing other reserved characters to the command processor without the meaning that is normally attached to them.

To illustrate this usage, let's look at an instance in which a space is used to separate two elements in a command line that are not separate arguments. For this example we will use the sort_list command, a rather uncomplicated command (fully described in the Multics WORDPRO Reference Guide, Order No. AZ98) which you may actually come to use quite often. The command's syntax is:

```
sort_list pathname -sort STR
```

The pathname is that of a segment which is designed specifically to hold lists, and the -sort control argument indicates how the segment is to be sorted. It does that by using, in the argument position marked by STR, one or more of the names that differentiate elements within the list, such as "lastname" and "firstname."

Now, assume that you have a segment in your directory that contains a list of customers' names and you want to sort that list by last name and first name. For instance, if you had both John Doe and Jane Doe in your list, you would want Jane placed before John in the alphabetically sorted list. In this case you must include both "lastname" and "firstname" in the argument to the -sort control argument. So, you might type:

```
! sort_list customers -sort lastname firstname
```

where "customers" is the pathname of the segment containing your list of customers.

But this invocation would return the error message:

```
sort_list:  Specified control argument is not implemented by
            this command.  firstname
```

because you are using a space between "lastname" and "firstname." Since the character string following the control argument is an argument in its own right, the space between "-sort" and "lastname" operates legitimately to separate the two. And when the command processor encounters the space between "lastname" and "firstname," it interprets "firstname" to be another argument. But the -sort control argument, takes only one argument, so the presence of what appears to be a second causes an error message to be returned.

In order to suppress the normal meaning of the space character here, and thereby make the two separate character strings appear as one argument, you must enclose the argument to the control argument in quotation marks:

```
! sort_list customers -sort "lastname firstname"
```

ITERATION

Iteration is one of several methods Multics provides for economizing typing of the command line. By enclosing elements of a command line in parentheses, you can have each of the elements processed separately. This enables a user to change one or more of the elements used in processing a command. For instance, if you wanted to print three segments with the print command you might type:

```
! print seg_1; print seg_2; print seg_3
```

But with iteration you could simply type the command once and then enclose the three segment pathnames in parentheses:

```
! print (seg_1 seg_2 seg_3)
```

Parentheses used in this fashion on Multics indicate that the individual items separated by blank spaces within the parentheses are to be processed separately by the command. With the command line used here, the segments would be printed one after another, starting at the left.

Iteration can also be used with command names. If you wish to invoke two commands on the same segment, you would type:

```
! (print delete) seg_1
```

In effect, this command line is:

```
! print seg_1; delete seg_1
```

Also, more than one iteration can be invoked with a command. In such a case, each element from one iterated set is processed with a corresponding element from another set. For example:

```
! rename >Smith_dir>(Jones Brown Doe) (Day White Green)
```

expands into:

```
rename >Smith_dir>Jones Day
rename >Smith_dir>Brown White
rename >Smith_dir>Doe Green
```

Iterated sets may also be nested, that is, placed one with the other. This practice is particularly useful when subsets of an element are repeated. Parentheses are evaluated from left to right. For example:

```
! create_dir >Smith_dir>(new>(first second) old>third)
```

creates three directories:

```
>Smith_dir>new>first
>Smith_dir>new>second
>Smith_dir>old>third
```

The directory names "first" and "second" are nested within the iteration composed of the two elements "new" and "old>third." The directory name "new" is added to ">Smith_dir>" first along with one of the elements from the nested iteration--"first." Since there is a nested iteration attached to "new," it is called a second time with the other element from the nested iteration--"second." Then the second element in the outer iteration--"old>third"--is added to the directory name ">Smith_dir>."

It is important to note here that there must be a blank space between separate elements in an iteration (e.g., "first" and "second") because they are separate arguments on the command line.

We have now developed some fairly detailed command techniques that demonstrate how precise Multics commands can be when applied with arguments and control arguments. In subsequent sections we will look at still more ways of specifying the manner in which Multics commands are to be run.

SECTION 3

ACTIVE FUNCTIONS

An active function is a program that provides current information such as the date, name of the day, and name of your last message sender. Some of the information supplied by active functions can be supplied by commands, and in fact many active functions can act as commands, just as some commands can serve as active functions. But the principal service of an active function is to place current information in a command line that is then executed with that information as a part of the command's program. When active functions are typed in a command line, their programs are executed first and the results of those programs returned to the command line to be processed with the command. It is in this way that you can make many standard Multics commands conditional; that is, the results of the invocation of the complete command line depend upon the information received from the active function.

To see the results of an active function you can type some on a terminal as commands. Not all active functions can be typed as commands (those that can are listed in the MPM Commands), but the two that follow are used to demonstrate what results from the invocation of an active function. The first example shows an active function which returns information by itself, that is, without using a pathname:

```
! date_time
  08/09/79      1130.2 est Wed
```

The second example uses an active function which requires a pathname argument for its operation:

```
! contents seg_1
  On a clear day you can see forever.
```

ACTIVE FUNCTIONS AS SUBSTRINGS

When an active function is used as an argument in a command line, the normal usage, it must be surrounded by brackets. The simplest form is:

```
[af arg1 ... argN]
```

where af is the name of an active function and argi represents the character string arguments to the active function. (Throughout Multics documentation, syntax lines that explain the structure of command lines containing a variable number of arguments use argument 1 (arg1) through argument N (argN), the N representing the number of the last argument used. Collectively, these arguments are referred to as argi.)

In the following example, an active function is included in a command line:

```
! send_message [last_message_sender] Thanks for the link.
```

When you type a newline and send this message to the command processor, the last_message_sender program is executed and the resulting value returned to the command line, in this case a User_id. The resulting value is not printed on the terminal, nor is the command line reprinted, but if it were it would look like this:

```
send_message JDoe.Pubs Thanks for the link.
```

That of course is just what would appear if you took the time to type the last message sender's User_id.

More than one active function can be used in a single command line. Suppose, for instance, that, wishing to keep track of the time you spend on individual jobs, you record the time at which you begin each job by sending yourself a message like the following:

```
! send_message [user name].[user project] Mailing list started at
  [date_time]
```

This command line uses the active function "user," which requires an argument (see the description of this active function in the MPM Commands). Here your Person_id and Project_id would be returned to the command line along with the date and time that you typed the message and it could all be saved in your mailbox as a message:

```
From Smith.Pubs 11/12/79 1200.7 est Mon:
Mailing list started at 11/12/79 1200.7 est Mon
```

Nesting Active Functions

Active functions can also be nested in a command line, that is, one function included within another. For instance, if you wish to underline the contents of a segment and have it printed on your terminal, you would use the string active function as a command and type:

```
! string [underline [contents seg_1]]
```

Of course, all executions within the brackets will be processed before the string command is processed. Each time the command processor encounters a right bracket (]) it returns to the matching left bracket ([) and evaluates the enclosed active function. This means that the innermost active function is evaluated first. To execute the above line, the command processor first evaluates [contents seg_1] and returns, for example:

```
On a clear day you can see forever.
```

Then the command processor evaluates the next element:

```
[underline On a clear day you can see forever.]
```

and returns:

On a clear day you can see forever.

The string command then operates on this string by printing it out on the terminal.

The term "active string" is often used interchangeably in Multics documentation with the term "active function." Actually, it refers to the entire string of characters enclosed within a single set of brackets, and this could include several separate active functions. Nesting, as we've just seen, is one way of including more than one active function within a set of brackets. Another way is to separate active functions with semicolons, as in the following:

```
! string [plus 3 4; times 5 6]
```

Here the active function string is used as a command on the active functions plus and times. The two active functions in this active string are processed separately and returned as arguments to the command. After the results of these two active functions have been returned, the command line is:

```
string 7 30
```

Since string used as a command simply prints the arguments that follow it, the printed result of this command line is:

```
7 30
```

Iteration of Active Functions

Active functions can also be used with iteration. (See Section 2 in this manual for a full description of iteration.) For example, if a segment named "all" contains the name of three segments, seg_1, seg_2, seg_3, then the command line:

```
! string [contents all]
```

will print the names returned by the active function contents:

```
seg_1 seg_2 seg_3
```

If, on the other hand, the active function was also included in parentheses:

```
! string ([contents all])
```

then the command would print the names of each of the three segments vertically:

```
seg_1
seg_2
seg_3
```

because the active function returns the three names within the parentheses so the command processed is:

```
string (seg_1 seg_2 seg_3)
```

In effect, the string command is being invoked three times in this command line, as if the line were:

```
string seg_1; string seg_2; string seg_3
```

Iteration can also be used within an active string, as in the following:

```
! string [(plus times) 2 3]
```

Here the two active functions are processed separately, each one using the arguments 2 and 3. Both active functions are processed before the command executes, so the command line becomes, in effect:

```
string [plus 2 3] [times 2 3]
```

This then becomes:

```
string 5 6
```

and the command execution causes the following line to be printed:

```
5 6
```

Rescanning

After an active string has been evaluated, the return value is rescanned for additional active strings before being inserted into the command line; the command processor continues scanning until all pairs of brackets have been evaluated. For example, if seg_4 contains just one line consisting of the string ([contents all]) described above, then the command line:

```
! string [contents seg_4]
```

invokes ([contents all]) as an active function which returns seg_1 seg_2 seg_3, the contents of the segment named "all," enclosed in parentheses so that the command is executed as:

```
string (seg_1 seg_2 seg_3)
```

or:

```
string seg_1; string seg_2; string seg_3
```

This rescanning can be stopped by placing a double vertical bar (||) before the active function. If, for instance, you want to print the contents of a segment which contains right and left brackets and not have those brackets interpreted as another active function, you would use the double vertical bar as it is used in the following command line:

```
! string ||[contents seg_5]
```

With this notation the entire returned value would go to the command processor without being checked again for brackets. Assume that the contents of seg_5 are:

```
Now [time of crisis] is the time for all good men to come to the
aid of their country.
```

Without the double vertical bar, this returned value would be rescanned. The word "time" would be read as a valid active function, but since "of crisis" is not a valid argument for this function, an error message would be printed on the terminal. But with the double vertical bar in front of the original active function brackets, as in the above command, the string command will simply print out the above segment, including the brackets and enclosed phrase.

There also may be times when you wish to rescan a returned value for some reserved characters (see Section 2 in this manual for a list of reserved characters) but not for brackets. In those cases, a single vertical bar (¦) will cause a returned value to be rescanned only for quotation marks. For example, consider the command line:

```
! string [contents seg_6]
```

where the contents of seg_6 are:

```
The symbol ] will be used to mark the end of each category.
```

After the first scanning of this command line, the returned value would be included in the line as:

```
string  The symbol ] will be used to mark the end of each
        category.
```

Since the rescanning procedure continues until all brackets have been dealt with, this line would be rescanned and an error message returned because there is no left bracket to match the right one.

But if the command line included a single vertical bar before the active function:

```
! string ¦[contents seg_6]
```

then the returned sentence would not be rescanned for reserved characters such as brackets. The command would be sent to the command processor and the sentence, with the right bracket included, would then be printed.

ACTIVE FUNCTION ERRORS

If the command processor detects an invalid input argument or some other error in an active function that prevents it from returning the expected value, it signals the active_function_error condition. The standard system action for active_function_error is to print a message describing the error that was found and create a new command level. For example, in the following command line, arguments that designate segments have not been included with the contents active function, so an error message has been returned saying why the active function cannot be processed:

```
!  string [underline [contents]]
   contents: Wrong number of arguments supplied

   Error: Bad call to active function contents
   r 14:03 0.090 76 level 2
```

You will note here that the command level is now level 2. Whenever another command level is created, the bad command line is held and you are shifted to a new command level. For this reason you should type:

```
!  release
```

in order to release the bad command or commands and return to the original level. Then you can correct the error and retype the command line.

Not all errors made with active functions signal the active_function_error condition. For instance, if you do not match brackets properly, you will get an error message, but you will not be placed to another command level:

```
!  string [underline contents seg_1]]
   command_processor_: Brackets do not balance.
   r 14:04 0.034 50
```

Since the command processor has returned you to the original command level, you need not type release. You need only type the corrected command.

SECTION 4

IMPORTANT COMMAND LANGUAGE FEATURES

Several features of Multics enable you to apply commands to a wide range of segments simultaneously. Thus far we have concentrated on the flexibility with which you can invoke single commands with a limited number of pathnames. In this section we will see how Multics command language enables you to multiply both the commands and the pathnames executed by issuing one command line. It is with these features that you gain truly efficient control over the large quantity of information that Multics allows you to store.

STAR NAMES

A detailed command like the one constructed in Section 2 can also be applied with precision another way--across a wide range of specified pathnames. One way to do this is by using the star and equals conventions. Many commands that accept pathnames as input allow any component of the final entryname in the path to be a star. This star (*) then represents all names that appear in that position of the pathname, and thus the command will operate on a range of pathnames rather than on just one.

The list command will serve well to illustrate this feature. Suppose you store in your working directory a number of segments whose entrynames have "plans" as their last component. With these you keep track of all your plans and changes you make in them. To list them all without using the star convention would require you to type each pathname in full:

```
!  ls seg_1.plans seg_2.plans new.plans rev.plans old.plans

   Segments = 5, Lengths = 8

   r w     1  seg_1.plans
   r w     3  seg_2.plans
   r w     1  new.plans
   r w     1  rev.plans
   r w     2  old.plans
```

With the star convention, you can get all of these segments listed with a much shorter command line:

```
!    ls *.plans
```

Use of this single asterisk can be expanded to fit a variety of situations like the following (the -no_header control argument is used to eliminate the list header from these examples):

```
!  ls seg_1.*.* -no_header

   r w    1  seg_1.new.plans
   r w    2  seg_1.old.plans
```

The above sample execution lists all segments with entrynames composed of three components, the first being "seg_1."

```
!  ls *.plans -no_header

   r w    1  add.plans
   r w    1  seg_1.plans
```

The example above lists all segments whose entryname has two components, the second (and last) being "plans."

```
!  ls *.*.* -no_header

   r w    1  seg_1.new.names
   r w    2  seg_1.old.plans
   r w    1  add.new.plans
```

The above example lists all segments with three-component entrynames.

Sometimes occasions arise when you wish to specify part of a component as variable rather than the whole component. Multics provides for this with the question mark (?) feature of the star convention. The question mark represents just one letter of an entryname. Suppose, for instance, that you wish to list all pathnames consisting of one three-letter component.

```
!  ls ??? -no_header

   r w   1  add
   r w   1  new
```

Suppose then that you want to list all segments in your working directory with two-component entrynames, the second containing the word "plans" followed by exactly one character. In this case you would use a question mark and a star name:

```
!  ls *.plans?  -no_header

   r w   2  seg_1.plans2
   r w   1  seg_1.plans1
   r w   1  add.plans1
```

These two features can even be used together in the same component of an entryname. In the following example the command will list all segments with one-component entrynames beginning with "ad" and containing at least three characters:

```
!  ls ad?* -no_header

   r w  1  add
   rew  2  additional
```

In this case the question mark and star are used together because the user wants listed only segments whose entrynames have at least three characters. The question mark cannot be interpreted as null, that is, having no corresponding character, so only entrynames with at least one character after "ad" are listed. But the star can be interpreted as null when it is used to represent part of a component, so entrynames with no more than one character after "ad" are listed (e.g., "add"). So too, the above command used without the question mark would list two-character entrynames as well as those with three or more characters:

```
!  ls ad* -no_header

   r w    1  add
   rew    2  additional
   rew    1  ad
```

Yet another feature of the star convention is the double star (**) which matches any number of components (including none) in the corresponding position in the entryname. For instance, the following command line will list all entries in the working directory which have "plans" for the last entryname component.

```
!  ls **.plans -no_header

   r w    1  seg_1.plans
   r w    1  add.plans
   r w    2  seg_1.old.plans
   r w    1  plans
```

Notice that this form of the pathname argument will also return an entry whose only component is "plans"; that's because the double star can be interpreted as null. In order to get only "plans" entries with two or more components, you would type:

```
!  ls *.**.plans -no_header

   r w    1  seg_1.plans
   r w    1  add.plans
   r w    2  seg_1.old.plans
   r w    1  add.new.plans
   r w    1  seg_1.old.test.plans
```

The single star is added here because it is not interpreted as null when it is used to represent an entire component. Thus, the star name in this command line returns only entrynames with at least two components. The single star can be interpreted as null only when it represents part of an entryname component (e.g., "ad*"), not when it represents an entire component.

The double star, on the other hand, can be interpreted as null whenever it is used, though of course there would be no reason to use it for representing anything but one or more complete entryname components.

EQUAL NAMES

Some commands that accept pairs of pathnames as their arguments (e.g., the rename command described in the MPM Commands) allow any component of the second entryname to be an equal sign. This equal sign (=) then represents the corresponding component of the first entryname given after the command name. For instance, if a segment named random.data is to be renamed ordered.data, the user would type:

```
rename random.data ordered.=
```

The convenience of this is more significant when several entrynames are being typed. For example, in the following add_name command:

```
add_name world.data =.statistics =.census
```

is equivalent to:

```
add_name world.data world.statistics world.census
```

The equal name convention (commonly referred to in Multics documentation as the equals convention) becomes extremely useful for matching series of entryname components when it is combined with the star convention. In the following command, all two-component entrynames with data_base as their second component are renamed with data as their second component:

```
rename *.data_base =.data
```

The combination of star with equal name can be extended, as in this case, for example, where you wish to rename all segments whose entrynames have "plans" as their last component. The command:

```
rename *.plans old_=.=
```

will append "old_" to each first component of the following entrynames:

```
program.plans
add.plans
seg_1.plans
```

On the other hand, the above command would not change the following segment names:

```
1.program.plans
new.add.plans
my.seg_1.plans
```

because the star name path "*.plans" returns only segments with two-component entrynames whose second component is "plans."

Another equal name feature that is comparable to a star name feature is the double equal sign. Like the double star, the double equal represents more than one entryname component, as in the following command:

```
rename one.two.three 1.==
```

which is equivalent to:

```
rename one.two.three 1.two.three
```

In this example the double equal sign stands for all components following the first component, in this case two components, "two" and "three."

In the example that follows, the entryname using an equal name contains more components than the matching entryname. Thus the double equal sign does not correspond to any components of the matching entryname and it is ignored. The commands:

```
rename alpha.beta ==.x.y
rename alpha.beta x.y.==
rename alpha.beta x.==.y
```

are equivalent to:

```
rename alpha.beta x.y
```

Like the double star, the double equal sign can be interpreted as null. In this example, only the specified components of the matching entrynames, "x" and "y", are used because two are enough to match the initial entryname, "alpha.beta."

There is a difference between the way single and double equal signs are interpreted when they have no corresponding component. When a single equal sign appears in a position where no corresponding entryname component exists, Multics responds with an error message, such as in the following:

```
rename alpha beta.=.gamma

rename: Illegal use of equals convention.
        beta.=.gamma for alpha
```

Unlike the double equal sign, the single equal sign cannot be interpreted as null. This usage is, therefore, illegal because there is no second component in the entryname "alpha" with which to match the equal name component. But with a double equal sign the command would function because this sign can represent any number of components, including none, as is the case in the three examples above where "alpha.beta" is renamed "x.y."

Finally there is the triple equal sign feature of the equals convention. The triple equal sign component represents all components of the corresponding entryname and thus no other component of the equal name may contain an equal sign. The triple equal sign is used to add components to a name, as in the following:

```
rename test.plans ===.old
```

which is equivalent to:

```
rename test.plans test.plans.old
```

There is one last aspect of the equals convention to discuss, though you may not use it very often. It is the percent sign (%), and it is similar to the question mark in the star convention. The percent sign (%) represents a single character in a specific position in the corresponding entryname component. For instance, the command:

```
rename ???*.data %%%.=
```

renames all two-component entrynames that have a last component of data and a first component containing three or more characters so that the first component is truncated to the first three characters and the second component is data (e.g., alpha.data would be renamed alp.data).

CONCATENATION

Yet another feature of Multics command language is concatenation, the practice of joining separate character strings together to form one string. When a character string bounded by reserved characters (often called a delimited element) is placed next to a string or another delimited element in a command line, the two are concatenated. You can thus form character strings by concatenating elements such as parenthetical expressions, active functions, and quoted strings. If, for instance, while working in another directory you wish to rename a segment in your home directory, you could concatenate the home_dir active function with the pathname arguments of the rename command to economically invoke the command:

```
rename [home_dir]>seg_1 seg_1.old
```

The home_dir active function would return the character string name of your home directory (e.g., >udd>Pubs>Smith). The command line shown here would thus change:

```
>udd>Pubs>Smith>seg_1
```

to:

```
>udd>Pubs>Smith>seg_1.old
```

HOW COMMANDS CAN BE INTERRUPTED

Often it is desirable to interrupt a command before its execution is complete. You may discover while the command is executing that a mistake has been made, or it may simply not be necessary to execute the command entirely. For example, you may issue the print command but not need to see the entire segment printed. So as soon as the needed information is printed, you could issue a quit signal. The quit signal's key varies from terminal to terminal; it may be either BREAK, BRK, INTERRUPT, INTRPT, or ATTN. The quit signal causes Multics to stop whatever it is doing and instead print QUIT and a ready message.

The ready message printed after a quit signal is slightly different from other ready messages because it contains additional information after the standard numbers:

```
r 9:38 1.123 62 level 2
```

The character string "level 2" indicates that a new command level has been established and the interrupted work is being held on the previous level. Since the system is at command level, that is, ready to accept more commands, you can either continue the interrupted work or go on to something else.

If the work interrupted by the quit signal is to be continued, you can issue either the start (sr) or the program_interrupt (pi) command. The start command resumes execution of the interrupted command from the point of interruption, and the program_interrupt command resumes execution of the original command from a known, predetermined reentry point. Usually the program_interrupt command is invoked when you are working in a subsystem like qedx or read_mail and you want to interrupt printing and remain in the subsystem. This method of resuming an interrupted command is useful for skipping over information not needed at the time. After the QUIT message is printed, typing the program_interrupt command will return you to request level.

If, on the other hand, you do not wish to continue the interrupted work, the interrupted command should be released before any other commands are issued. It is expensive to hold interrupted commands at a command level. The release command (rl) releases the work interrupted and held by the quit signal and returns the system to the previous command level (and drops the level information from the ready message).

SECTION 5

ABBREVIATION AND ARGUMENT SUBSTITUTION

There are on Multics several commands that enhance your ability to type command lines efficiently. Two of these commands--abbrev and do--will be discussed in this section, and a third--exec_com--will be discussed in Section 6.

THE abbrev COMMAND

The abbrev command enables you to create your own abbreviations for the elements you use in command lines. For instance, if you found yourself repeatedly changing to another working directory with a command like the following:

```
! cwd >udd>Training>Jones
```

you could create an abbreviation for the pathname in order to avoid typing the lengthier form. The command line might then be as short as:

```
! cwd J
```

The letter "J" would be expanded to the character string it represents, in this case ">udd>Training>Jones," and the command would then process with this pathname.

You create abbreviations by invoking the abbrev command and then using a command-like invocation called an abbrev request line. To illustrate this first step, let's look at how you would create the abbreviation for ">udd>Training>Jones." First, you invoke the abbrev command:

```
! abbrev
  r 10:30 1.321 64
```

When invoked, the abbrev command returns no output: you simply get a ready message. But once invoked the abbrev command remains in effect until you cancel it. That is done by issuing an abbrev request line:

```
!  .q
```

You may seldom find it necessary to issue this quit request. In fact, the abbrev procedure is so useful for minimizing typing at the terminal that users often include the abbrev command in their start_up.ec (see Section 6 in this manual) so that it is automatically invoked every time they log in. It is while the abbrev command is in effect that you can create, delete, and change abbreviations and use them in command lines.

To create an abbreviation you must type a request line, which begins with a period (.) in the first non-blank space. To create an abbreviation that will be expanded no matter where it appears in a command line, you use the control request .a:

```
   r 10:30 1.321 64

!  .a J >udd>Training>Jones
   r 10:31 0.011 65
```

This request line places the abbreviation "J" in a special segment that is labelled with your Person_id and a suffix of profile (e.g., Smith.profile). All of your abbreviations are stored in this segment, unless you specifically place some in other segments. You may, if you wish, have several separate segments for abbreviations. You would then specify with the abbrev request ".u" which group of abbreviations you'd like to use at that time.

For the sake of simplicity, we'll assume here that all abbreviations are being placed in one segment named Smith.profile. So far, we've placed one abbreviation in this segment:

```
J = >udd>Training>Jones
```

Now it can be used as an abbreviation anytime the abbrev command is in effect.

When it is invoked, the abbrev command sets up a special processor, called the abbrev processor, which works on each command line input to the system. This processor scans each line to detect and expand abbreviations and then passes the command line on to the normal command processor. In this process abbreviations are expanded only once, so you cannot nest abbreviations.

To continue our illustration of the abbrev command feature on Multics, let's look at another type of abbreviation, one which is expanded only when it appears at the beginning of a line. Such a specific entry would be useful for the abbreviation of an entire command line or a part of the line that includes the command name. You might do this with the command line used above to change your working directory. In order to indicate that this abbreviation is to be expanded only when it appears at the beginning of a command line, type:

```
!  .ab C cwd >udd>Training>Jones
```

Thus, to change to this particular directory, you would type only:

```
!  C
```

You'll no doubt find need for a number of abbreviations, and to keep track of what they are, you'll occasionally want to read your profile segment. To do this while the abbrev command is in effect, you simply type:

```
!  .l
```

You will then have a list like the following printed at your terminal:

```
      J     >udd>Training>Jones
      S     Smith
      N     new.plans
      A     add.plans
      O     old.plans
      arc   art_customers
  b   C     cwd >udd>Training>Jones
```

The first six abbreviations shown here will be expanded if they appear anywhere in the command line. The lower case "b" to the left of the last abbreviation indicates that it will be expanded only when it appears at the beginning of a command line.

Any abbreviation you create must be no more than eight characters long, and when you type it into a command line it must be bounded by break characters. And of course this latter condition makes it impossible for the abbreviation itself to contain any break characters. The characters that the abbrev processor treats as breaks are:

formfeed
vertical tab
horizontal tab
dollar sign ($)
apostrophe (')
grave accent (`)
period (.)
less than (<)
greater than (>)
braces ({})
parentheses (())
brackets ([])
newline
space
quotation mark (")
semicolon (;)
vertical bar (|)

You will notice that the last seven characters in this list are those that were called reserved characters in Section 2. It is important to remember that while these characters are being used for the special purposes described earlier, they will also serve as break characters and thus possibly set off any abbreviations you are using. To be on the safe side, you should only use break and reserved characters when you have a specific need for them. That should prevent the expansion of characters by the abbrev processor when you do not mean them to be interpreted as abbreviations.

In cases where you want to use characters that are defined as abbreviations for some other purpose, you can prevent the abbrev processor from expanding them by enclosing them in quotation marks. For instance, if you want to change working directories using a pathname that contains the entryname arc, you couldn't type:

```
cwd >udd>Training>arc
```

Because the string arc is defined as an abbreviation in your profile segment, this pathname would be expanded to:

```
>udd>Training>art_customers
```

The command would probably not be able to find a directory by that expanded name and thus would return on error message:

```
change_wdir: Some directory in path specified
does not exist.  >udd>Training>art_customers
```

But this expansion could be suppressed by quotation marks:

```
cwd >udd>Training>"arc"
```

It could also be suppressed by the request .<space>:

```
. cwd >udd>Training>arc
```

By beginning the command line with a period and a space, you suppress expansion of the entire line, that is, no abbreviations contained in the line will be expanded.

An effective way of avoiding unanticipated expansions is to use capital letters in abbreviations. Since Multics command language uses lower case letters, it is very unlikely that any string you use from command language will ever be confused with an abbreviation if your abbreviations use upper case letters. For instance, the entryname "arc" could not be confused with the abbreviation for "art_customers" if the latter used a capital A--"Arc."

Your existing abbreviations are also checked when you are adding abbreviations. If an abbreviation you are creating already exists, you will be asked whether or not you actually do wish to redefine it. You simply respond "yes" or "no".

THE do COMMAND

The do command enables you to substitute arguments in a command line before executing the line. This is particularly useful for command lines that repeat a certain argument a number of times, such as those containing multiple commands.

Let's return to the segments containing plans that we used in previous illustrations. Suppose you are about to compose a new set of plans and want to discard your old plans. You decide to print a copy of the segment old.plans before deleting it, and you want to rename your most recent plans, currently in the segment named new.plans, to old.plans. To do this you could type:

```
! print old.plans; delete old.plans; rename new.plans old.plans
```

But with the do command you could avoid retyping the segment names by referring to them with special symbols included in the command line:

```
! do "print &2; delete &2; rename &1 &2" new.plans old.plans
```

Numbers preceded by an ampersand (&) refer to arguments listed after the quoted portion of the do command line. In the example, "new.plans" is substituted for the &1 string at each point where &1 appears because "new.plans" is in the first position after the quoted portion of the command line. Likewise, "old.plans" substitutes for &2 because it is in the second position after the quoted string, separated from the first by a blank space. So, the above command line would expand to the following when the do command is executed:

```
print old.plans; delete old.plans; rename new.plans old.plans
```

This, of course, is the command we originally wanted to type.

It is important to note here that if an argument is not supplied, nothing will be inserted in the places where the extra number and ampersand appears. The last argument would not be used to substitute for the extra places. For instance, if the command line shown above had an &3, it would be ignored:

```
! do "print &2; delete &3; rename &1 &2" new.plans old.plans
```

would be expanded to:

```
print old.plans; delete ; rename new.plans old.plans
```

After the print command executed, the delete command would return an error message because it has not been supplied with a pathname:

```
Usage:  delete paths -control_args
r 13:08 0.186 53
```

If, on the other hand, the command could execute regardless of the empty argument, it would not return an error message; it would go ahead and execute.

The do command is particularly useful in conjunction with the abbrev command. Earlier it was noted that abbreviations cannot be nested because the abbrev processor scans a command line only once. But the do command makes, in effect, two command lines out of one, so the abbrev processor does scan the command twice in this case, though of course it will not expand any abbreviations within the quoted command string during the first scan. You might, for example, use "P" as an abbreviation for "plans" in the command line shown above and thus type:

```
! do "print &2; delete &2; rename &1 &2" new.P old.P
```

The "P" would first be expanded:

```
do "print &2; delete &2; rename &1 &2" new.plans old.plans
```

and then substituted, producing the expansion shown earlier:

```
print old.plans; delete old.plans; rename new.plans old.plans
```

Often-used do command lines can even be added to your profile segment, a practice that will enable you to make very long command strings very easy to type. Take as an example one of the lines typed above and create an abbreviation for it, using the additional abbreviation "P" just suggested:

```
! .ab PLAN do "print &2; delete &2; rename &1 &2"
```

This will then reduce typing of the above string with new.plans and old.plans as the do command arguments to:

```
! PLAN new.P old.P
```

This expands first to:

```
do "print &2; delete &2; rename &1 &2" new.plans old.plans
```

and then the do command is executed, producing:

```
print old.plans; delete old.plans; rename new.plans old.plans
```

Because the do command uses quotation marks, it is necessary that we now understand further the convention of quoted strings in the Multics command language. We noted in Section 2 that quotation marks are used for passing characters exactly as they are typed on the terminal, that is, suppressing the interpretation normally applied to them. As we have just seen above, they can be used in this manner to suppress the expansion of an abbreviation. But when one set of quotation marks is included within another set of quotation marks, as is quite likely with the do command, the inner quotes must be doubled.

For instance, if you apply the do command to the sort_list command line used as an example in Section 2:

```
sort_list customers -sort "lastname firstname"
```

it becomes:

```
! do "sort_list &1 -sort ""lastname firstname""" customers
```

The inner quotation marks must be doubled because characters are interpreted individually from left to right. Thus the single quotation mark followed by another type of character (e.g., "sort_list &1 -sort "lastname...) would be interpreted as the end of the quoted string that began with the first quotation mark. In that case, the quoted portion of the do command line would appear to be "sort_list &1 -sort" and everything following it would be interpreted by the command processor as arguments for substitution. That, of course, is not the intention at all.

But as it is in the above command line, the doubled quotation marks are interpreted as single marks (") because they are enclosed within the outer quotation marks of the do command line. So, after the do command substitutes the argument, the command line is what we had in the original example:

```
sort_list customers -sort "lastname firstname"
```

because the double quotation marks have been reduced to single marks by the do command.

SECTION 6

exec_com

The exec_com command offers yet another means of abbreviating the typing involved in command invocation. With exec_com you can place frequently used command sequences in segments that are then processed by the invocation of the exec_com command. Plus it offers the feature of control statements, which permit more variety and control in the execution of command sequences. This procedure enables you to invoke a large number of commands with only one command and arguments.

This feature is made even more flexible by the inclusion of arguments to the command that can be substituted for special strings in the exec_com segment. By this means, and of course by including active functions as well, you can have the entire sequence of commands act on different input each time it is executed. And to deal with variations in the execution process that different input might necessitate, you have the advantage of control statements.

CREATING AN exec com SEGMENT

An exec_com segment is created with a text editor and can make use of any of the Multics command conventions. The entryname you assign the segment must have the suffix ec (e.g. print.ec).

To illustrate the creation and functioning of an exec_com segment, we will create a short, simple segment comprised only of commands, that is, without any control statements. To change your working directory, print it, and list its segments and your access to them, you would type in the following lines with a text editor such as qedx (described in Part I):

```
! qx
! a
! change_wdir &1
! print_wdir
! list
! \f
! w change.ec
! q
```

Argument Substitution

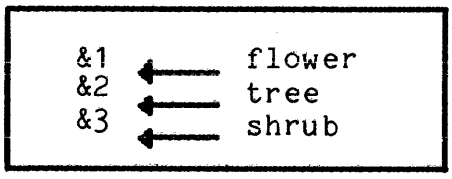
The ampersand character (&) and number used in the change_wdir command line refers to an optional argument that is substituted when the exec_com command is invoked, just as is done with the do command. The ampersand character is also used in exec_com to signify the start of a control statement; that will be discussed below. The particular values that are to be substituted are placed on the exec_com command line, as in the following:

```
! exec_com change.ec >udd>Training>Jones
```

When you are working with an exec_com segment that calls for more than one argument to be substituted, you arrange the arguments in sequence, separated by blank spaces, after the pathname argument on the command line. Let's suppose you have a segment named action.ec which requires three arguments. The command line used to execute this segment would look like the following:

```
! exec_com action.ec flower tree shrub
```

The three arguments--flower, tree, and shrub--would be substituted for ampersands and numbers in the following order:



If the third argument were not supplied on the command line:

```
! exec_com action.ec flower tree
```

The space occupied by &3 would be left blank; the second argument "tree" would not be substituted in that space. If possible, the commands, active functions, and control statements will process without the missing arguments, but if they cannot, an error message will be returned. Often, but not always, the error message will tell you which part of the exec_com segment is not functioning.

There are also some special substitutions that can be made in exec_com segments. First of all, you can, if you wish, place the number of optional arguments supplied with a particular execution into the exec_com lines. In the action.ec segment used above, any position containing the figure &n would receive the number 3 before the segment was executed.

In another substitution you can place the entryname portion of the exec_com's pathname, without the ec suffix, into the exec_com segment. In the action.ec segment, any position containing the figure "&ec_name" would receive the entryname action before the segment was executed.

Lastly, you can place the directory name portion of the exec_com's pathname in the exec_com segment by using &ec_dir.

Control Statements

Control statements enable you to specify conditions for command execution and transfer execution to different parts of the exec_com segment. Currently there are twelve control statements:

&label and &goto

&attach, &detach, and &input_line

&command_line, &ready, and &print

&quit

&if, &then, &else

As the list indicates, ampersands signify the start of a control statement. Normally, control statements must start at the beginning of a line without any leading blanks. However, a &then can be on the same line as an &if, and other control statements can follow either &then or &else on the same line. For instance:

```
&if...
&then  &goto...
```

or

&if... &then &goto...

Some control statements set conditions for the ensuing input or execution of commands while others alter the normal sequence of command execution. An instance of the former is the &print statement. The execution of this statement causes the input which follows it to be printed at your terminal.

To illustrate this type of control let's look at the execution of another exec_com segment, called query.ec:

```
change_wdir [response "Working directory desired?"]
&print  Your working directory is:
print_wdir
&quit
```

When you invoke the exec_com command on this segment:

```
!  exec_com query.ec
```

you will get the following output:

```
change_wdir [response "Working directory desired?"]

Working directory desired?  !  work
Your working directory is:
print_wdir
>udd>Pubs>Smith>work
```

You will notice that, in addition to the things this segment is designed to print, all of the segment's command lines are reprinted in the output. This can be prevented by another statement that establishes the conditions for the execution of subsequent commands, the &command_line off control statement. As a matter of fact, this control is almost always used in exec_com segments because the command lines seldom need to be reprinted. If we were to place &command_line off at the beginning of query.ec, invocation would produce:

```
Working directory desired?  !  work
Your working directory is:
>udd>Pubs>Smith>work
```

The &quit control statement is another of those that set conditions in that it marks the end of the exec_com segment. It is good practice to include the &quit statement because later you may want to place several exec_coms together in one exec_com segment. In those cases it will be necessary to mark clearly where one exec_com ends and another begins.

Then there are those control statements that alter the normal sequence of execution, the most obvious of which is the &if statement. This is used primarily with the &then and &else statements, and by using other control statements with these, you can further enhance the versatility you have to deal with varying situations. For instance, by including a &goto statement with a &then statement, you can skip over commands and go to a specific location when conditions specified by &if exist.

To understand this type of control statement, let's look at an illustration which uses it. The following exec_com segment is more intricate than our earlier examples because it combines command lines, a variety of control statements, and active functions, and it requires argument substitution.

```
&command_line off
&if [compare &1 &2]
&then print &1
&else &goto process
&quit
&label process
&if [compare &2 &3]
&then print (&1 &2)
&else print (&1 &2 &3)
&quit
```

After turning off printing of its command lines, this exec_com tests two segments for likeness with the compare command used as an active function. The two segments to be compared are supplied as arguments in the exec_com command line. The compare active function returns the word "true" if the contents of both segments are the same and the word "false" if the contents are different. The control statement &if operates on the word that is returned. When the word "true" is returned, the &if statement shifts control to the &then control statement. When the word "false" is returned, the &if statement shifts control to the &else control statement. (The &if control statement is similar to the if command in that a then statement is required but an else statement is not.)

The clause following &then, which must be on the same line, can include a command line, an optional argument, and the null statement. It can also include other control statements, except &label, &if, &then, and &else. These conditions are the same for the clause following &else.

Because the only action desired when the compare active function returns true is that one of the segments be printed, the clause following &then is simply the print command and a pathname argument. The &else condition, on the other hand, requires a further switching of control because it involves more than one command or control statement. Thus you use a &goto statement with a name, in this case "process," which automatically switches the point of execution to the &label statement with the matching name. Execution then resumes at the line immediately following &label.

Notice here that the line following &else is &quit. This statement causes the current invocation of exec_com, that is, all the subsequent command lines, to cease, which is exactly what you want when the condition is true and the &then statement has been executed. If &quit were not included, execution would go to the second &if statement even when the original condition of the compare active function proved true.

The second &if control statement in this sample exec_com is designed to compare the segment supplied as the second optional argument with a third segment. If the compare active function returns "true" here, the &then clause, using the print command with iteration, has the first two segments printed at your terminal. When the comparison of the second and third segments returns "false," control shifts instead to the &else control statement, which has all three segments printed.

start_up.ec

The start_up.ec is a special exec_com segment that contains commands to be executed each time you log in, before anything is read from your terminal. In fact, you do not even need to invoke the exec_com command for this segment; it is automatically invoked as part of the login procedure. The only things necessary for this automatic invocation are that the segment be named "start_up.ec" and that it reside in your home directory.

This feature of the exec_com is useful because users usually have certain operations that they want performed almost every time they log in. With a start_up.ec they are saved the work of typing the required commands each time they enter the system. And if ever they wish to log in without executing these commands, they simply add the control argument -no_start_up (-ns) to login.

There are several commands that almost all users include in their start_ups. These include:

abbrev
: so that you can use your personal abbreviations during each terminal session

print_motd
: so that the system prints the message of the day when you haven't seen it before

print_mail
: so the system automatically prints your mail

accept_messages
: so you receive messages from other users online

Then, of course, you probably would want to include the &command_line off control statement at the beginning of your start_up so that the segment's commands aren't printed at your terminal each time.

There are also quite a few other commands that it is useful to invoke when you start up, and we will discuss some of the most important ones. To take advantage of your terminal's capabilities and make typing easier, you may want to include several set_tty commands in your start_up on a conditional basis. This command provides many options and is invoked through the usage:

```
set_tty -modes OPTION1,OPTION2,OPTION3...,OPTIONn
```

Note that with the control argument -modes there is the unusual use of commas between its character strings, but there are no spaces.

Ordinarily, several set_tty commands are used conditionally in the start_up.ec so that you have the proper one actually invoked for the terminal you are using at a particular log in. Thus you would have a set of &if &then control statements for each type of terminal you might log in from. An example of the form is:

```
&if [equal [user term_type] TN300]
&then set_tty -modes ll118,crecho,lfecho
```

The user term_type active function would return your particular terminal type and if it proved, by the active function equal, to be the TermiNet 300, then the important modes would be set appropriately.

In the -modes control argument above, the ll118 sets the line length at the TN300 to 118 columns or spaces. Without this specification your terminal would be set at 79 columns, the default setting. You could also use a number other than 118, depending on the limitations of a particular terminal. The crecho (carriage return echo) designation creates a situation in which a carriage return is provided each time a linefeed is typed. The lfecho (linefeed echo) designation provides for one linefeed each time you hit the carriage return key. Thus you can get a newline by typing either a carriage return or a linefeed. You must note, however, that the TermiNet 300 has an automatic linefeed switch which, when turned on, provides a linefeed automatically each time a carriage return is typed. So if you have that switch on and have the lfecho mode set, you will get a double linefeed.

In order to gain an in-depth understanding of how start_up.ecs work and what they are used for, let's look at a somewhat complicated but quite realistic example of one. For the sake of explaining this example, the lines are numbered, though of course they couldn't be in the actual exec_com.

```
 1.  &command_line off
 2.  &if [equal &2 interactive] &then &goto interactive
 3.  &else send_mail [user name][user project] Absentee
         run started at [date_time]
 4.  &goto all
 5.  &label interactive
 6.  &if [equal &1 new_proc] &then &goto new_proc
 7.  &else print_motd
 8.  check_info_segs
 9.  &if [have_mail] &then string "You have mail."
10.  &label new_proc
11.  set_tty -modes polite
12.  accept_messages -print
13.  &if [equal [user term_type] TN300]
14.  &then set_tty -modes ll118,crecho,^lfecho
15.  &if [equal [user term_type] ASCII]
16.  &then set_tty -modes crecho,lfecho
17.  &if [equal [user term_type] VIP7801]
18.  &then set_tty -modes crecho,lfecho
19.  &label all
20.  abbrev
21.  &quit
```

This start_up.ec handles three conditions: logging in as an interactive user, which is what you ordinarily do, running an absentee job (discussed in Section 7) and creating a new process once you're already logged in.

Line 2 checks to see if you are logging in as an interactive user, and if you are, sends control to line 5. The &2 argument is supplied automatically by the Multics system. When the start_up.ec is invoked, the system places either the word "interactive" or the word "absentee" in the second optional argument position so one of those words is supplied in the &2 location; and here it is compared by the equal active function to the character string "interactive." If the active function returns false, because the invocation is for an absentee process, then execution begins at line 3, which sends a message to your mailbox telling you what date and time your absentee job began running.

Line 4 then takes control and sends execution to the &label statement at line 19. The word "all" is an appropriate &label name here because the lines following 19 are always executed when the start_up.ec is invoked, whether it be for interactive, absentee, or new process use. Those lines invoke the abbrev command and mark the end of the exec_com segment.

If the check for interactive usage in line 2 had proved true, then you would be going through a different execution sequence before getting to lines 19 through 21. In such a case, control would, of course, skip over lines 3 and 4, go to line 5 and start executing at line 6.

Line 6 checks to see if you are creating a new process at this point in your interactive usage. A new process is created by the new_proc command or by an error that cancels your process. The new_proc command cancels the current process and sets up a new one, using the control arguments given initially with the login command and the optional argument to the new_proc command itself. It's as though you logged out and immediately logged back in. The start_up.ec is again invoked automatically when a new process is started. The optional argument the system automatically gives in the first position with the invocation of the start_up.ec is the character string login or new_proc, depending on whether the invocation is coming from the initial log in or from the new_proc command. So in line 6, &1 is substituted with one of those strings. If that string is new_proc, then the equal active function returns true and execution is shifted to line 10, &label new_proc.

What the &goto control statement in line 6 passes over are executions that would have been accomplished at the initial interactive log in and would not need repeating. Line 7 provides for a special handling of the message of the day. If you have no start_up, the message is printed each time you log in; but when you once place a start_up.ec in your home directory, the system assumes that you are taking action on your own to examine the message and thus stops printing it automatically. When incorporated in your start_up, the print_motd command keeps a copy of the last message of the day in a segment (called Person_id.motd) in your home directory. Each time the command is invoked, usually during execution of your start_up, it compares the current message with the saved one and prints the current one if it is changed from the saved copy. This way you don't have to see the message again and again when once is enough.

The check_info_segs command in line. 8 is handy to have in the start_up because it prints a list of new or modified segments in the info segment library each time you log in. Much like the print_motd command, it controls the listing by saving the current time in the user's profile so that when it is invoked again, it lists only info segments created or modified since the last invocation.

Line 9 checks your mail for you each time you log in interactively. By making the procedure conditional, as is done here, you can avoid the full printing of all your mail that you would get if you simply used the print_mail command. This way, nothing is printed if you have no mail, and only "You have mail." is printed if you do. When this latter is the case, you can then type the read_mail command, which allows you a great deal of control over printing the messages, after the start_up has finished executing.

Provided you are not running an absentee process, your start_up executions will eventually get to line 10, whether from the &goto in line 6 or by executing through line 9. In either case, you will then pass on to lines 11 through 18 and then, of course, go on from 19 through 21, as in all start_up invocations. What happens here is first, in line 11, an unconditional setting of terminal modes to polite. Polite holds the printing of any output sent to your terminal (e.g., messages from other users) while you are typing input until the carriage returns to the left margin (i.e., when you type a newline).

The accept_messages command in line 12 allows your process to accept messages sent by the send_message command and notices of the form "You have mail." sent by the send_mail command. The -print control argument prints all messages sent to you by the send_message command since the last time you were logged in and accepting messages. Messages and mail notices sent while you are logged in will then be printed out at your terminal, in this case when you next return the carriage to the left margin because you have the polite mode on. This is necessary because the check for mail, performed by line 9 in this start_up, does not check for messages sent by the send_message command.

If, while you are logged in, you defer messages with the defer_messages command, all mail and messages sent to you are saved in your mailbox and can be read with the read_mail or print_messages commands.

The only lines in this start_up that we haven't discussed thus far are 13 through 18. These 6 lines contain three tests of terminal types, the procedure explained above using TN300 as an example. The three included in this start_up are three that a user is very likely to encounter--TN300, ASCII, and VIP7801. But this start_up.ec would not prevent its user from logging in on another terminal. That terminal's default modes would simply be in force because no special modes are set by the start_up. Of course, the user could easily set modes for that terminal by using the set_tty command while logged in.

The only convention used in these six lines that was not explained earlier is the ^lfecho mode. The circumflex character (^) acts as a negation, turning the particular mode off. This start_up uses ^lfecho because the TermiNet 300 has the automatic linefeed switch.

Because it provides a way of storing oft-used command sequences, an abbreviated way of invoking them, and an internal means for controlling their execution, the exec_com command is one of the most powerful and useful features of the Multics system. And there are yet other aspects of the feature which can enhance your efficiency on the system. There are facilities for adding search paths to the exec_com search list, for answering questions generated by exec_com sequences, for combining a number of exec_coms into one segment, and for handling conditions raised during execution of an exec_com. In fact, there are ways that you can call one exec_com from another, or reenter the current one at an earlier point. These additional features are explained in full in the MPM Commands.

SECTION 7

ADDITIONAL CONCEPTS

The purpose of this section is to explain several concepts that are important in computer technology and have somewhat specialized applications in Multics. Some of these concepts you've probably encountered already and others you no doubt soon will. The presentation here will stay mainly on the conceptual level, that is, with no explanation of the procedure for implementing the processes. Any applications demonstrated will be used simply for the purpose of illustrating how a concept is applied to a procedure.

ONLINE

Being online means being logged in, entering information and using information already stored in the system. In Multics, online is an interactive process in that the system responds immediately to the user's input. This way you do not have to prepare an entire job beforehand to be run all at once; instead, the system will interact with you.

A necessary additional aspect of interactive usage is time-sharing. Many users can interact with the system at the same time, even to the point of sharing the same segments simultaneously. For this reason, the system tells you each time you log in how many people are online and how many the system can accommodate at that time. And because the computer is shared this way, you may sometimes be preempted while working on the system.

ABSENTEE

Absentee, on the other hand, is a process that can be run when the user is not logged in and interacting with the system. Instead, the user prepares an entire job beforehand and has it run at a specified time. It is analogous to batch processing on other systems. And an absentee job is not a time-sharing process in the sense that online processes are. The absentee job is placed in a waiting line (a queue) and run as background to the normal interactive work on the system.

The principal difference between an absentee process and an interactive one is that in an absentee process the I/O switches are attached to special absentee segments instead of to a terminal. One of these segments is the control segment containing commands and other input data which you create with a text editor. The other is an output segment which stores the results of the absentee job. The system adds a third component to the User_id to distinguish absentee from interactive processes: absentee processes are labelled Person_id.Project_id.m while interactive processes use an "a" as the third component. These third components are called "instance tags."

The details of executing absentee processes are given with the enter_abs_request command in the MPM Commands.

Punched cards, when they are run on Multics, are processed in a batch similar to the manner in which absentee jobs are processed. The standard way of handling card decks in Multics is to place the deck in the card reader and read it into a system pool. You then log in on a terminal and transfer the card file from the system pool to your working directory using the copy_cards command. The segment that this command creates is stored in the system can be used in interactive and absentee processes, just as a segment created on a terminal would be used.

STORAGE SYSTEM

The segment is the basic unit of storage in the Multics system. It can vary in size, that is, in the amount of information it contains, and it may contain a collection of program instructions, text or other data, or it may be empty (a null segment). There is a limit to the amount of information that can be stored in a segment, but if any single collection of information is too long for one segment, it can be stored in a group of segments called a multisegment file.

Multics keeps track of segments by cataloging them in directories. The base directory, the one from which all other directories and all segments emanate, is called the root directory. Figure 7-1 uses a representation of an inverted tree to demonstrate the relation of user Tom Smith and his project, Pubs, to the root. (Directories are represented by rectangles and segments by lozenges.) Notice the two directories immediately under the root (sss and udd). The sss (system_library_standard directory) is one of several library directories that catalog all the system commands and subroutines. The udd (user_directory_directory) directory is a catalog of project directories. It contains one directory entry for each project on the system. Likewise, each project directory normally contains one directory for each user on that project.

The Multics system's virtual memory makes all segments in the storage system directly addressable. That means that in effect there is hardly any difference between main memory and secondary storage on Multics: information can be retrieved from storage virtually as fast as from memory.

Since the physical movement of information between secondary storage and main memory is totally automatic, its structure is of no concern to the user when working on a process. A user does not have to be concerned with where and on what devices the segments reside.

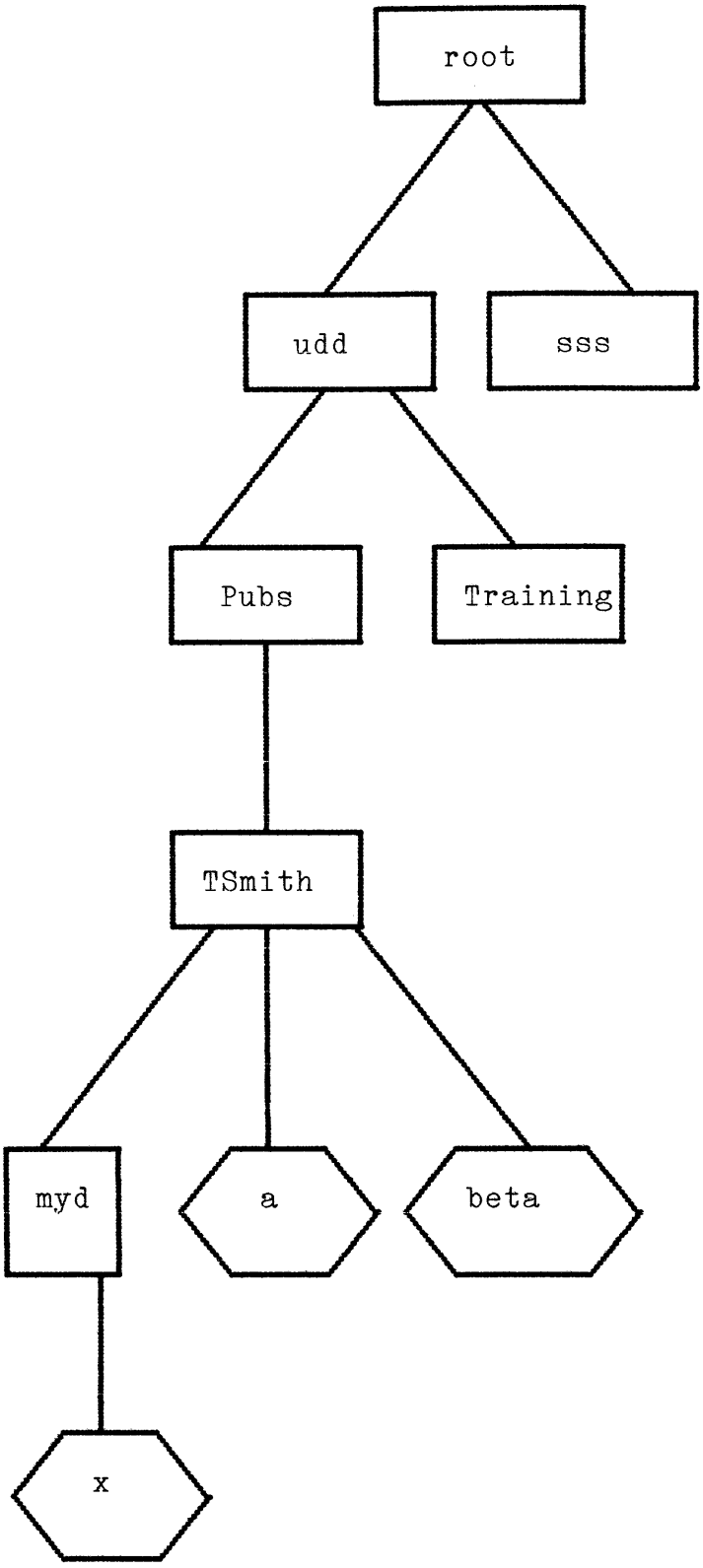


Figure 7-1. Hierarchical Storage System

SEARCH RULES

Whenever the user issues a command or references a program or other segment, the system must search through directories to find the specified command or program. This search is regulated by a list of search rules that specify a set of directories to be searched in a particular order. But the search rules that the system automatically follows may be changed or supplemented by the user. The set_search_rules command enables the user to change the default search rules, and the add_search_rules and delete_search_rules commands enable the user to add or delete search directories. To check your current search rules, you can invoke the print_search_rules command.

Adding another directory to be searched after the working directory is a convenient way for an entire project to share a group of special programs peculiar to the work of that project. After a user on the project adds this special directory to the search rules, any programs in that directory can be executed as easily as the system commands. This addition to the search rules means that each user on the project saves the time and cost of either copying each one of the programs or linking to each one.

Also, by manipulating the search rules, the user can determine whether a system command or a user-written command with the same name is to be used.

LINKING

Multics allows a user to create a link to a segment anywhere in the storage system, as long as access to the directory of the linked segment is available. By creating a link, you can reference another segment as though it were in the directory containing the link. In short, you can use this particular segment without actually having to make a copy of it.

Linking with the link command is not to be confused with dynamic linking. The latter is the Multics term for the mechanism in the system that provides a highly efficient means of referencing stored segments. It is enough, for our purposes here, to say that, with dynamic linking a segment must be searched for only once during a process. When a segment is found by using the search rules, its place in the storage system is remembered so that another search does not have to be made the next time the segment is needed.

BOUND SEGMENTS

A bound segment is a single executable segment made up of two or more separately compiled segments. Normally, you would bind program segments that you intend to execute together repeatedly. (See the bind command in MPM Commands.) Bound segments are easier for the user to process, and the system can run them much more efficiently than under the regular dynamic linking procedure. In fact, by binding segments you can not only save execution time, you can also save money through decreased computing time and storage space.

ARCHIVE SEGMENTS

Each segment in Multics is assigned space in increments of pages. Since this can result in quite a bit of blank space on the last page of segments, Multics provides the archive command to pack the contents of individual segments together into one archive segment. You can maintain control over these individual segments by invoking the archive command with different arguments, (see the archive command in MPM Commands). The advantage of archive segments is that they reduce the user's storage load and therefore cost.

EDITOR MACRO

In general computer terminology, macro refers to a group of executable statements. To that extent a macro is like an exec_com segment on Multics. But in Multics the term macro is applied only to a sequence of text editor requests. This sequence of requests, called an editor macro, acts like a program or exec_com. Placing it in a separate buffer preserves it for repeated application in the text editing environment.

APPENDIX A

GLOSSARY

The following list of terms is meant to add to the glossary provided in Part I of this New Users' Introduction (CH24). Most of the terms appear for the first time here, though several are repeated with expanded definitions.

absolute pathname
see pathname, absolute

character string
One group of characters unbroken by blanks; it signifies one word to Multics. The characters may include alphabetic, numeric and some other characters (periods, hyphens, and underscores).

command level
The state the computer is in when it is ready to accept command lines. You are at command level when you log in, when a command completes execution or encounters an error, or when you stop command execution by issuing a quit signal. Command levels above level 1 are indicated by the ready message.

command processor
The program that interprets command lines and calls the appropriate programs, after processing parentheses and active functions.

component (entryname)
A part of an entryname. Entryname components are separated by a period (e.g., data_base is the second component of the entryname random.data_base.plans).

crash (FNP)
an unplanned termination of service from the front-end network processor causing a disconnection of the process. The process can be saved and reconnected when the -save_on_disconnect control argument has been used with the login command.

crash (system) a
An unplanned termination of system availability caused by problems in hardware and/or software, often signalled by the message: MULTICS NOT IN OPERATION. Processes cannot be reconnected after a system crash.

data base manager
A software system that integrates various computerized information units of an organization into a total system. With such a system, all users of data within an organization share common records of information and the information available at every level is drawn from the same source, providing mutually consistent levels of accuracy to all users.

default
The value or action that the system assumes when none has been specified by the user.

entryname
A name given to an item (segment or directory) contained in a directory. It may contain one or more components, separated by periods. All names given to entries within one directory are unique but need not be different from names used in other directories.

I/O switch
A path in the I/O system through which information is sent. For example, the normal output switch (user_output) is usually attached to the terminal, but it may be attached to a segment in storage by using the file_output command. This would save the output in a segment rather than print it at the terminal.

multiplexer
A communications control device which permits sharing of facilities by connecting the central processing unit to a large number of communications channels that may all transfer data to or from the processor at one time.

page (also known as record)
A unit of storage in Multics. A page contains up to 4096 characters.

pathname
A name of a segment or directory that specifies its location in the storage system. A pathname is either absolute or relative.

pathname, absolute

A segment name preceded by the series of directory names that lead from the root to that segment: each level in the pathname is preceded by a ">". For example, the absolute pathname for a segment under a user's home directory is designated this way:

>udd>Project_id>Person_id>segment_name

All absolute pathnames begin with ">".

pathname, relative

The pathname that uniquely locates a segment relative to the working directory, by listing the pathnames of directories under which the segment resides. For example, the relative pathname for a segment that resides in a directory one level under the working directory is designated this way:

lower_dir>segment_name

All relative pathnames begin WITHOUT ">".

process

The activities (programs, data entry, etc.) of an individual user that begin when the user logs in, including absentee log in, and continue until logout or until another process is explicitly begun through use of the new_proc command.

quota

The maximum number of pages that can be used in a hierarchy of directories. Each user is allotted a predetermined amount of quota; however, quota can be increased by a system administrator.

ring structure

The structure of access control on Multics which is implemented by special hardware. Operation is controlled in such a way that the computer's work is done in a number of mutually exclusive subsets. These subsets may be considered concentric rings of privilege, representing different levels of access rights. The innermost or hardcore ring is made up of those segments essential to all users. This innermost ring, designated as ring 0, represents the highest level of privilege. The work of most users is done in ring 4. Ring 7 is the ring of least privilege.

search rules

The rules that specify the order in which directories are searched to find a command, subroutine, or data item. This is to be distinguished from addressing a segment by its pathname, which explicitly specifies the directory containing the segment.

subsystem
: A collection of programs that provide a special environment for some particular purpose, such as editing, calculation, or data management. It may perform its own command processing, file handling, and accounting.

suffix
: The last component of an entryname, which often specifies the purpose of a segment (e.g., action.ec where ec specifies an exec_com segment).

APPENDIX B

FUNCTIONAL BREAKDOWN OF SELECTED MULTICS COMMANDS

Even as a new user you will fast find need for many more commands than have been discussed in Parts I and II of this New Users' Introduction to Multics. For that reason, this appendix provides you with a functional listing of Multics commands that are likely to become useful to you.

The categories here are similar to those used in other Multics documentation (MPM Commands and Multics Pocket Guide). Ten of the seventeen categories normally used in the other manuals are used here along with a category which lists Multics' word processing commands. Some commands appear in more than one category just as they do in the functional groupings in other Multics manuals.

Each category includes a description of how the commands in that group function. A complete description of the individual commands, except those used in word processing, is contained in the alphabetical listing of commands in MPM Commands. The word processing commands are described individually in the Multics WORDPRO Reference Guide (Order No. AZ98).

SELECTED COMMANDS LISTED BY FUNCTION

Access to the System

Access commands connect the terminal to a process. The hello command is more correctly called a preaccess request because it is used before a process has been set up; it repeats the greeting message that is printed whenever a terminal is first connected to the system.

dial
hello
login
logout

Storage System, Creating and Editing Segments

The commands in this category enable the user to create, edit, and format segments.

create
emacs
program_interrupt
runoff
edm
indent
qedx
runoff_abs

Storage System, Segment Manipulation

The commands in this category enable the user to compare and sort segments and adjust their sizes. Principally, they enable the user to manipulate a segment as a whole, copying it, truncating it, and moving it around in various ways in the storage system.

archive
copy
link
sort_seg
unlink
compare
delete
move
truncate

Storage System, Directory Manipulation

The commands in this category enable the user to create, manipulate, and delete directories.

copy_dir
delete_dir
list
rename
unlink
create_dir
link
move_dir
status

Storage System, Access Control

This category contains the commands that set, check, copy, delete, and list access to segments and directories.

check_iacl
copy_iacl_dir
delete_acl
delete_iacl_seg
list_acl
list_iacl_dir
set_acl
set_iacl_seg
copy_iacl
copy_iacl_seg
delete_iacl_dir
list_accessible
list_not_accessible
list_iacl_seg
set_iacl_dir

Storage System, Address Space Control

Commands in this category enable the user to manipulate search paths and search rules and the working directory. This category also contains the new_proc command, which creates a new process with a new address space; this is equivalent to logging out and logging back in.

add_search_paths
change_default_wdir
delete_search_paths
get_system_search_rules
list_ref_names
print_default_wdir
print_search_paths
print_wdir
set_search_rules
where_search_paths
add_search_rules
change_wdir
delete_search_rules
initiate
new_proc
print_proc_auth
print_search_rules
set_search_paths
where

Command Level Environment

Basically, commands in this category are designed to set up and control the environment in which other Multics commands are given. They enable the user to set the procedures by which other commands search for segments and directories, to manipulate the working directory, and to group commands on a command line or in a segment (even to abbreviate commands).

abbrev
add_search_rules
change_default_wdir
delete_search_paths
do
general_ready
if
memo
on
print_search_paths
print_translator_search_rules
program_interrupt
ready_off
release
set_search_rules
stop_run
add_search_paths
answer
change_wdir
delete_search_rules
exec_com
get_system_search_rules
line_length
new_proc
print_default_wdir
print_search_rules
print_wdir
ready
ready_on
set_search_paths
start
where_search_paths

Communication Among Users

The commands in this category give users facilities to send, receive, and store mail and short messages interactively with other users registered on Multics.

accept_messages
delete_message
print_auth_names
print_messages
send_mail
send_message_acknowledge
send_message_silent
defer_messages
immediate_messages
print_mail
read_mail
send_message
send_message_express
who

Communication with the System

With the commands in this category users can ask Multics what helpful information about the system operations is available on info segments and request help from particular info segments. Users can also find out how many people are using the system and who they are; in addition, users can request that the system's message of the day be printed at their terminal.

check_info_segs
how_many_users
print_motd
help
move_abs_request
who

Control of Absentee Computations

The commands in this category enter, move, and cancel requests to have work submitted by the user run by the system in the user's absence.

cancel_abs_request
enter_abs_request
how_many_users
move_abs_request
runoff_abs
cobol_abs
fortran_abs
list_abs_requests
pl1_abs
who

Wordprocessing

Commands in this category are used to create, edit, and format text and process lists of information.

add_symbols
change_symbols
copy_list
create_list
delete_dict_words
emacs
find_dict_words
list_dict_words
locate_words
option_symbols
print_wordlist
qedx
revise_words
sort_list
trim_wordlist
add_dict_words
compose
count_dict_words
create_wordlist
delete_symbols
expand_symbols
find_symbols
list_symbols
merge_list
print_symbols_path
process_list
retain_symbols
show_symbols
trim_list
use_symbols

APPENDIX C

FUNCTIONAL BREAKDOWN OF SELECTED ACTIVE FUNCTIONS

Like the commands, Multics active functions can be categorized by their operational use. The categories here are similar to those used in other Multics documentation (MPM Commands and Multics Pocket Guide). Ten of the thirteen categories used in the other manuals are used here. And some active functions appear in more than one category just as they do in the functional groupings in other Multics manuals.

Each category includes a description of what the active functions in that group do. A complete description of the individual active functions is contained in the alphabetical listing of active functions in MPM Commands.

REFERENCE TO ACTIVE FUNCTION BY GROUPS

Arithmetic

This group of active functions perform some arithmetic operation and returns the character string representation of the result. This group includes:

ceil	divide
floor	max
min	minus
mod	plus
quotient	times
trunc	

Character String

This operational group returns the results of various operations on one or more character strings. This group includes:

after	copy_characters
index	length
low	lowercase
reverse	underline
upper_case	

Condition Handling

The on active function is the only one in this group. It executes a command line and returns true if any of a particular set of conditions is signalled during the execution. If none of the specific conditions are signalled, it returns false.

Date and Time

This group consists of active functions that return information about the date and time in various forms. This group includes:

date	date_time
day	day_name
hour	long_date
minute	month
month_name	time
year	

Logical

This group returns a character string value of either true or false. Active functions in this group are intended to be used with the &if control statement of the exec_com command or with the if command. This group includes:

and	equal
exists	greater
less	nequal
ngreater	nless
not	or

Miscellaneous

These active functions return miscellaneous information about the user's process or storage system entries. This group includes:

contents	default

Pathname Manipulation

Active functions in this group construct a pathname based on the specified path argument and return all or part of this name. This group includes:

directory	entry
equal_name	path
strip	strip_entry
suffix	

Question Asking

This group returns a value based on the answer given by a user in response to a specified question. This group includes:

query	response

Storage System Names

These active functions return either pathnames or entrynames of existing entries. This group includes:

default_wdir	directory
entries	files
home_dir	links
segments	

User/Process Information

Active functions in this operational group return user information obtained from system data bases. This group includes:

have_mail	last_message
last_message_time	last_message_sender
severity	system
user	

INDEX

A

abbrev command 5-1, 6-7, 6-9, B-3

abbrev processor 5-4, 5-7

absentee 6-9, 6-11, 7-2, A-3, B-4

accept_messages command 6-7, 6-9, 6-11, B-4

active functions
  applied
    compare 6-5
    contents 3-1, 3-3, 3-4, 3-5, 3-6, 3-7
    date 6-5, 6-6
    date_time 3-1, 3-2, 6-9
    equal 6-5, 6-6, 6-9, 6-10
    greater 6-5, 6-6
    have_mail 6-9, 6-10
    home_dir 4-8
    last_message_sender 3-2
    plus 3-3, 3-4
    response 6-4
    string 6-5, 6-6
    times 3-3, 3-4
    underline 3-3, 3-7
    user 3-2, 6-9
  definition 3-1

add_name command 4-5, B-2

add_search_rules command 7-5

ampersand 5-6, 6-2, 6-3

archive command 7-6, B-2

archive segment 7-6

B

bind command 7-6

bound segment 7-6

break characters 5-4

C

card reader 1-2, 1-4, 1-5, 7-2

central processing unit 1-2, 1-3, 1-4, 1-5, 1-6, A-2

change_wdir command 2-6, 5-1, 5-4, 6-2, 6-4, B-3

character string 2-6, 2-7, 4-8, 4-9, 6-7, 6-9, 6-10, A-1, C-2

check_info_segs command 6-9, B-4

command level 3-7, 4-9, A-1, B-3

command processor 2-1, 2-2, 2-5, 2-6, 2-7, 3-2, 3-3, 3-5, 3-6, 3-7, 5-2, 5-5, A-1

commands
  applied
    abbrev 5-1, 5-2, 5-3, 5-7, 6-9
    accept_messages 6-9, 6-11
    add_name 4-5
    change_wdir 2-6, 5-1, 5-4
    check_info_segs 6-9, 6-10
    delete 2-8, 5-6
    do 5-6, 5-7
    exec_com 6-2, 6-4, 6-5, 6-6, 6-9
    help 2-2, 2-5
    list 2-2, 2-3, 2-4, 4-2, 4-3, 4-4
    new_proc 6-10
    print 2-1, 2-6, 2-8, 5-6
    print_messages 2-5
    print_motd 6-9, 6-10
    print_wdir 2-2, 2-5, 6-4
    program_interrupt 4-9
    release 4-9
    rename 2-8, 4-5, 4-6, 4-7, 5-6
    send_mail 6-9
    send_message 3-2
    set_tty 6-8, 6-11
    sort_list 2-7, 5-8
    start 4-9
    who 2-5
  definition 2-1
  system 2-2
  user-written 2-2

compiler 1-6

concatenation 4-8

control statement 6-1, 6-2, 6-3, 6-4, 6-5, 6-6, 6-7, 6-10, C-2

copy_cards command 7-2

CPU
  see central processing unit

crash (FNP) A-2

crash (system) A-2

D

data base managers A-2

default 2-2, 6-11, 7-5, A-2

defer_messages command 6-11, B-4

delete command 2-8, 5-6, 5-7, B-2

delete_messages command B-4

delete_search_rules command 7-5

do command 5-1, 5-5, 5-6, 5-7, 6-2, B-3

dynamic linking 7-5, 7-6

E

editor macro 7-6

enter_abs_request command 7-2, B-4

equal name
  see equals convention

equals convention 4-1, 4-5, 4-7
  percent sign 4-7

exec_com command 5-1, 6-1, B-3

F

file_output command A-2

FNP
see front-end network processor

FNP crash
see crash (FNP)

front-end network processor 1-4, 1-5, 1-6, A-2

H

hardware 1-2, 1-4, 1-5, 1-6, A-2, A-3

help command 2-2, 2-5, B-4

I

I/O switch 7-2, A-2

if command 6-6, B-3

input/output multiplexer 1-4, 1-5

interactive 6-9, 7-1, 7-2, B-4

IOM
see input/output multiplexer

iteration 2-8, 3-4

K

keypunch 1-2, 1-5

L

line printer 1-3, 1-5

link command 7-5, B-2

linking 7-5

list command 2-2, 2-4, 4-2, 4-3, 4-4, 6-2, B-2

login A-2

login command B-1

M

macro
see editor macro

main memory
see memory

memory 1-2, 1-3

multiplexer A-2

N

newline 2-1, 2-6, 3-2, 6-8

new_proc command 6-10, A-3, B-2

O

online 7-1

P

page 1-2, 7-6, A-2

peripherals 1-2

peripherals (cont)
  card reader 1-2, 1-4, 1-5, 7-2
  keypunch 1-2, 1-5
  line printer 1-3, 1-5
  storage devices 1-3
  terminal 1-2, 1-4, 1-5, 6-7, 6-8, 6-11, A-2, B-1

print command 2-1, 2-6, 2-8, 4-8, 5-6, 5-7, 6-6

print_mail command 6-7, 6-10, B-4

print_messages command 2-5, 6-11, B-4

print_motd command 6-7, 6-9, 6-10, B-4

print_search_rules command 7-5

print_wdir command 2-2, 2-5, 6-2, 6-4, B-3

process 7-2, A-2, A-3, B-1, B-2, C-3

program_interrupt command 4-9, B-2, B-3

Q

question mark
  see star convention

QUIT
  see quit signal

quit signal 4-8, 4-9, A-1

quota A-3

quoted strings 2-6, 4-8, 5-7

R

ready message 1-2, 2-5, 2-6, 4-8, 4-9, 5-1

read_mail command 6-11, B-4

release command 4-9, B-3

rename command 2-8, 4-5, 4-6, 4-7, 5-6, 5-7, B-2

reserved characters 2-6, 3-6, 4-8, 5-4

ring structure A-3

root directory 7-3, A-3

S

search rules 7-5, A-3

secondary storage
  see storage system

semicolon 2-5, 2-6, 3-3

send_mail command 6-9, 6-11, B-4

send_message command 3-2, 6-11, B-4

set_search_rules command 7-5, B-3

set_tty command 6-7, 6-8, 6-9, 6-11

software 1-2, 1-3, 1-6, A-2

sort_list command 2-7, 5-8, B-5

star convention 2-4, 4-1, 4-2, 4-4, 4-5
  question mark 4-3

star name
  see star convention

start command 4-9, B-3

start_up.ec 5-2, 6-7, 6-8,
    6-9, 6-10, 6-11

storage system 1-5, 1-6, 7-2,
    7-4, 7-5, A-2, B-2, C-3
  memory 1-2, 1-3, 7-3
  secondary storage 1-2, 1-3,
      7-3
  virtual memory 1-3, 7-3

string active function
  as command 3-3, 3-4, 3-6,
      3-7, 6-9

subsystem A-4

system crash
  see crash (system)

T

terminal 1-2, 1-4, 1-5, 5-7,
    6-7, 6-8, 6-11, A-2, B-1

time-sharing 7-1

V

virtual memory
  see storage system

W

who command 2-5, B-4

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: NEW USERS' INTRODUCTION TO MULTICS – PART II

ORDER NO. CH25-00

DATED NOVEMBER 1979

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy**: 32 Via Pirelli, 20124 Milano
Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F. **Japan**: 2-2 Jinbou-Cho Kanda, Chiyoda-Ku Tokyo
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia**: Mandarin Plaza, Tsimshatsui East, H.K.

30877, 5C183, Printed in U.S.A.

CH25-00